



Flight Software (FSW) Seminar



Aadil Rizvi
Brian Campuzano

*Jet Propulsion Laboratory
California Institute of Technology*

Apr. 19-20th, 2017

JPL Flight Software (FSW) Seminar

Special thanks to **Rajeev Joshi**

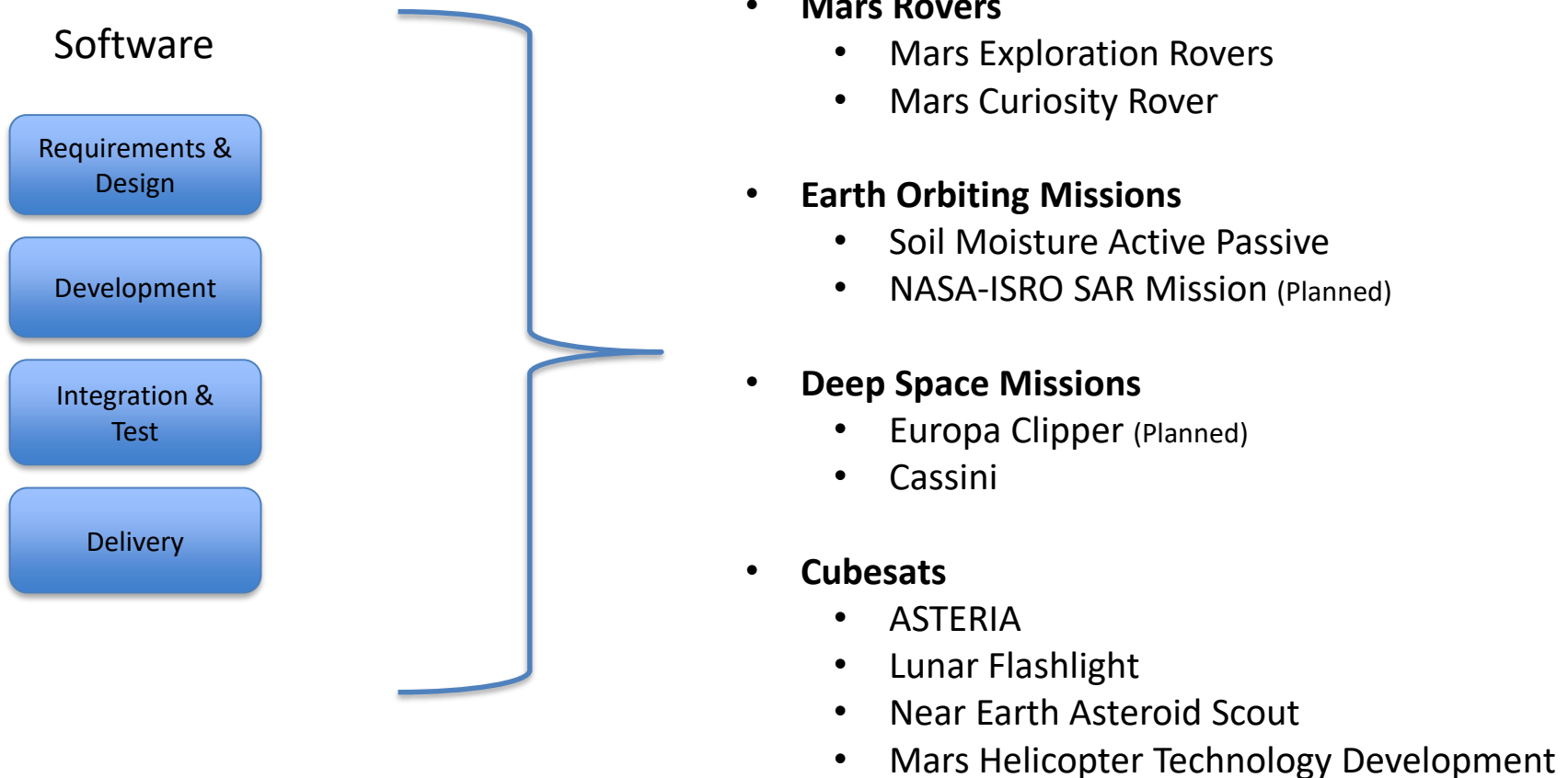


- Chief Engineer for Flight Software and Avionics Systems
- Laboratory for Reliable Software (LaRS)
- Member of flight software development teams for several JPL missions including: Europa Clipper, Mars 2020 Rover and Mars Science Laboratory
- Most seminar material is provided from his “CS Theory in Practice” course taught as part of the Flight Software Certification series at JPL

Agenda

- Flight Software and Avionics Systems (Section 348) Overview
- Intro to Flight Software (FSW)
- Flight Software Services
 - Commanding
 - Telemetry
 - Uplink
 - Downlink
 - Health
- Execution Platforms
 - Hardware
 - No Operating System / Bare Metal
 - Real-Time Operating System (RTOS)
- Threads
 - Concurrent Programming
 - States
 - Scheduling
- Scheduling Schemes
 - Preemptive scheduling
 - Priority Preemptive scheduling
- Thread Synchronization
 - Interference
 - Atomic Operations
 - Race Conditions
 - Shared Resources
 - Mutual Exclusion
 - Interrupt Locking
 - Semaphores
- Mars Pathfinder Priority Inversion Problem
 - What Happened
 - Priority Inversion Explained
 - Handling Priority Inversion
 - Programming Considerations with Semaphores
- The Producer-Consumer Problem
 - Scenario/Limitations Explained
 - Shared Bounded Queue
 - Using 1 semaphore
 - Using 3 semaphores
- Message Passing
 - Thread communication
 - Example application in a state machine
 - Issues in Message Passing
 - Pure Message Passing
 - Sharing a Resource with Message Passing
- The Sol 200 Anomaly

Section 348: Flight Software and Avionics Systems



Real-Time Embedded Software

Agenda

- Flight Software and Avionics Systems (Section 348) Overview
- **Intro to Flight Software (FSW)**
- Flight Software Services
 - Commanding
 - Telemetry
 - Uplink
 - Downlink
 - Health
- Execution Platforms
 - Hardware
 - No Operating System / Bare Metal
 - Real-Time Operating System (RTOS)
- Threads
 - Concurrent Programming
 - States
 - Scheduling
- Scheduling Schemes
 - Preemptive scheduling
 - Priority Preemptive scheduling
- Thread Synchronization
 - Interference
 - Atomic Operations
 - Race Conditions
 - Shared Resources
 - Mutual Exclusion
 - Interrupt Locking
 - Semaphores
- Mars Pathfinder Priority Inversion Problem
 - What Happened
 - Priority Inversion Explained
 - Handling Priority Inversion
 - Programming Considerations with Semaphores
- The Producer-Consumer Problem
 - Scenario/Limitations Explained
 - Shared Bounded Queue
 - Using 1 semaphore
 - Using 3 semaphores
- Message Passing
 - Thread communication
 - Example application in a state machine
 - Issues in Message Passing
 - Pure Message Passing
 - Sharing a Resource with Message Passing
- The Sol 200 Anomaly

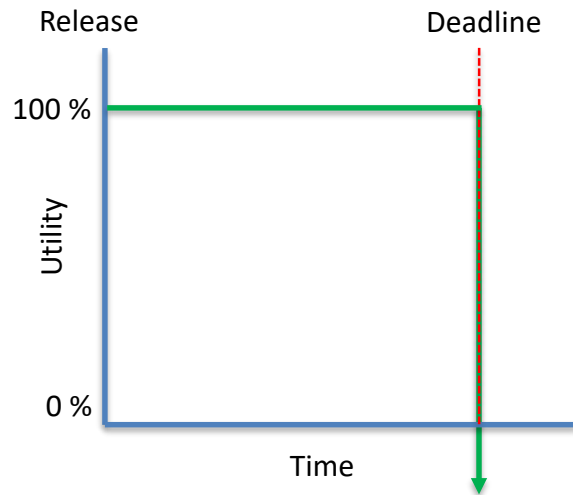
Intro to Flight Software (FSW)

- How is FSW different from any other software application?
- Hard real-time vs. soft real-time

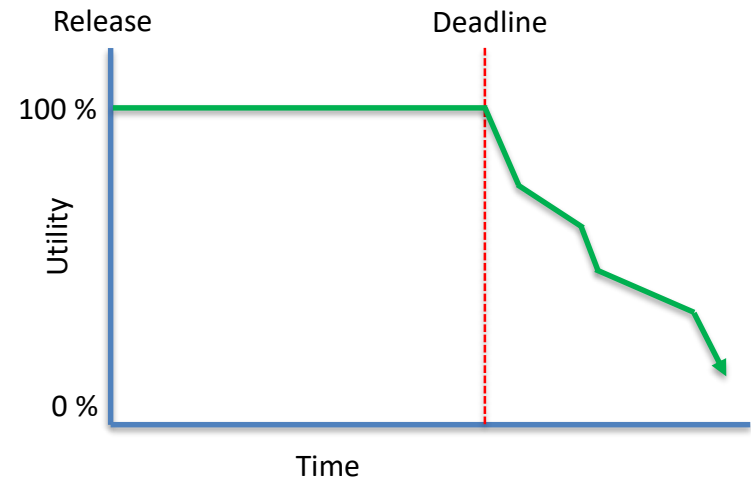
Intro to Flight Software (FSW)

A **real-time service** provides a transformation of inputs to outputs in an embedded system to provide a function

Hard Real-Time
Service Utility Curve



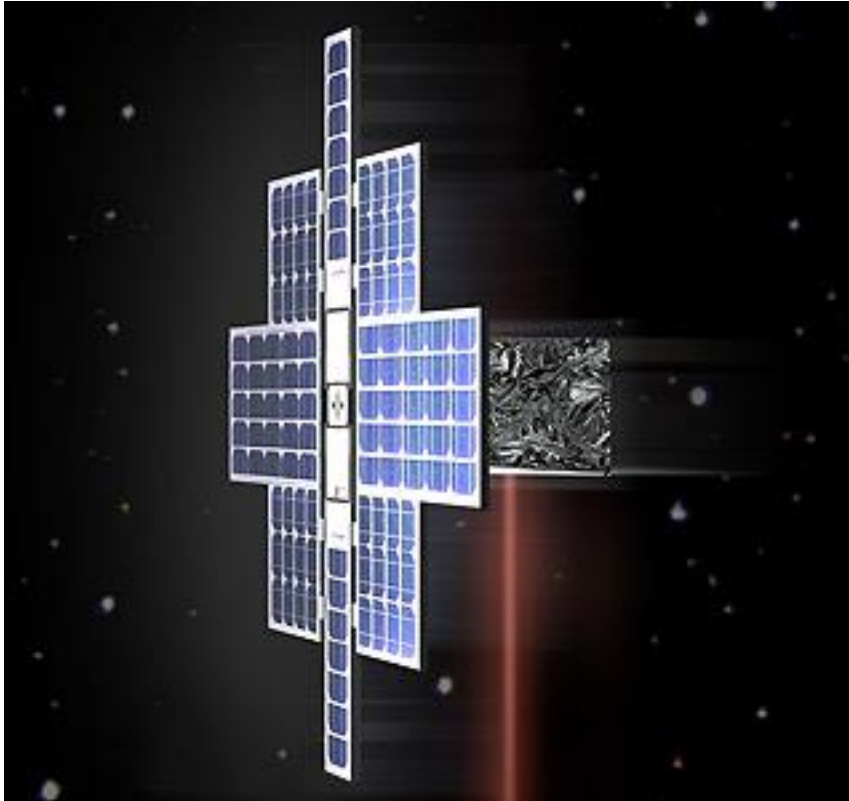
Soft Real-Time
Service Utility Curve



Agenda

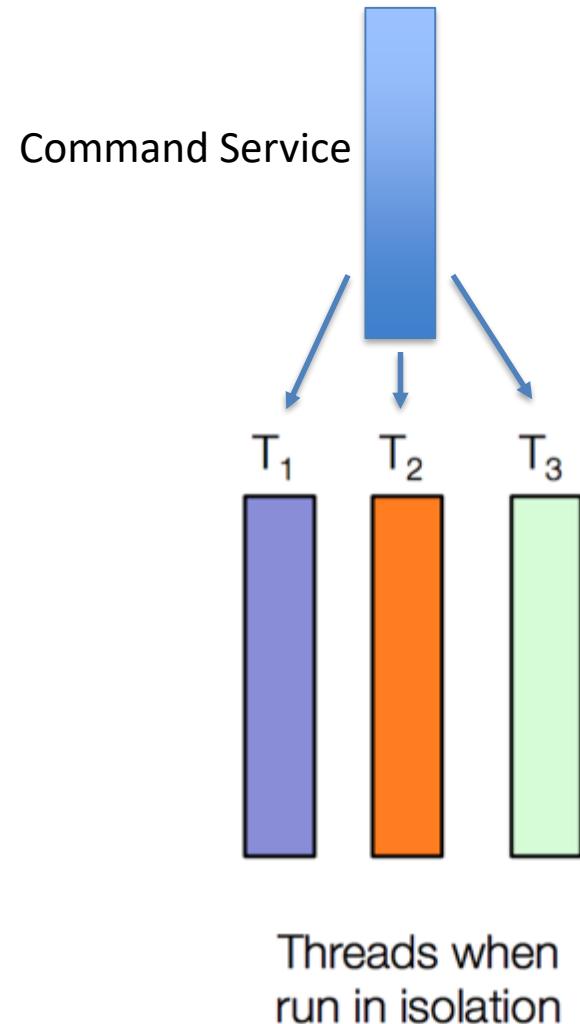
- Flight Software and Avionics Systems (Section 348) Overview
- Intro to Flight Software (FSW)
- **Flight Software Services**
 - Commanding
 - Telemetry
 - Uplink
 - Downlink
 - Health
- Execution Platforms
 - Hardware
 - No Operating System / Bare Metal
 - Real-Time Operating System (RTOS)
- Threads
 - Concurrent Programming
 - States
 - Scheduling
- Scheduling Schemes
 - Preemptive scheduling
 - Priority Preemptive scheduling
- Thread Synchronization
 - Interference
 - Atomic Operations
 - Race Conditions
 - Shared Resources
 - Mutual Exclusion
 - Interrupt Locking
 - Semaphores
- Mars Pathfinder Priority Inversion Problem
 - What Happened
 - Priority Inversion Explained
 - Handling Priority Inversion
 - Programming Considerations with Semaphores
- The Producer-Consumer Problem
 - Scenario/Limitations Explained
 - Shared Bounded Queue
 - Using 1 semaphore
 - Using 3 semaphores
- Message Passing
 - Thread communication
 - Example application in a state machine
 - Issues in Message Passing
 - Pure Message Passing
 - Sharing a Resource with Message Passing
- The Sol 200 Anomaly

Flight Software Services



- Commanding
- Telemetry
- Uplink
- Downlink
- Rate Groups
- Instrument/HW Interfaces
- Health Monitoring
- Spacecraft Time
- Fault Protection
- Memory/Data Management

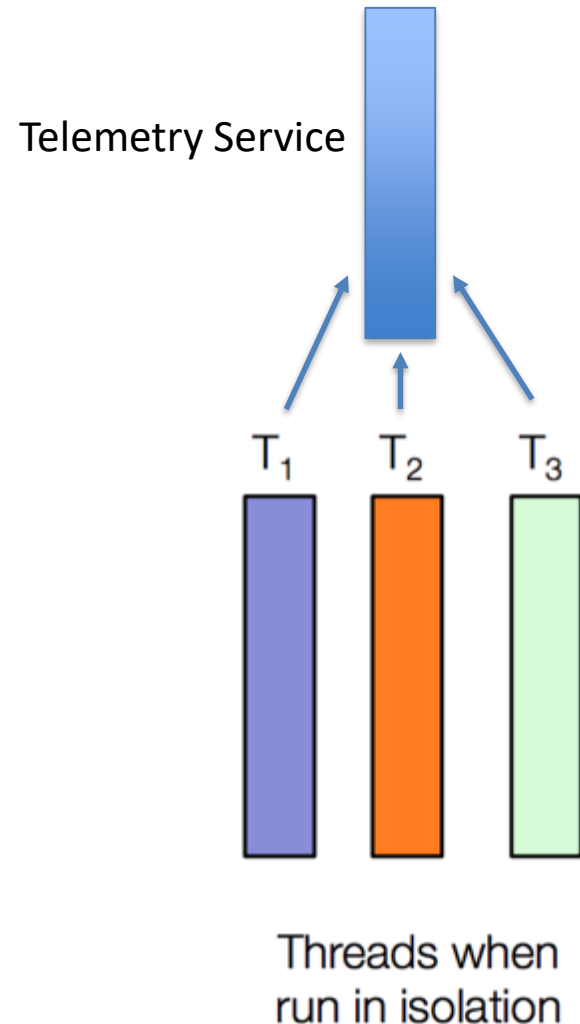
Commanding



- Individual components register command opcodes at initialization
- Upon receiving a command, the command task dispatches it to the target service for execution
- Command target service sends a response back to the command task upon completion

How is data passed from one task to another?

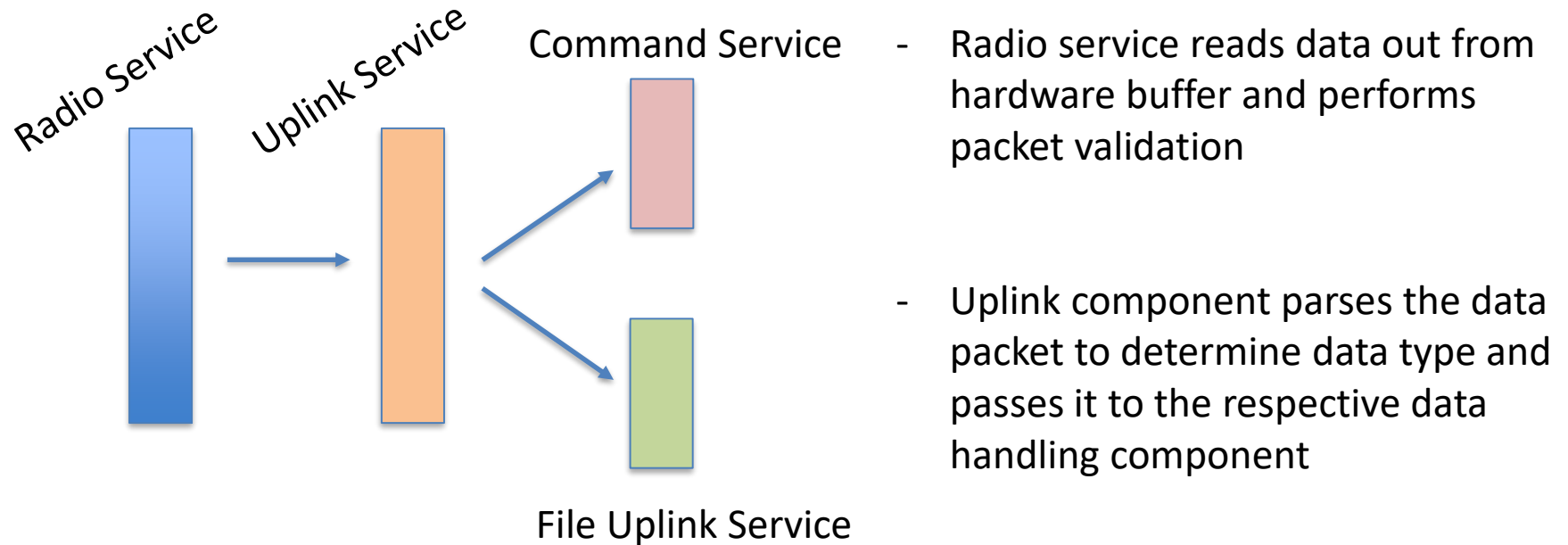
Telemetry



- Individual components assigned a range of telemetry IDs
- Components report telemetry/events to the telemetry/logging service
- Telemetry task reads reported telemetry entries and packetizes them for downlink and/or storage

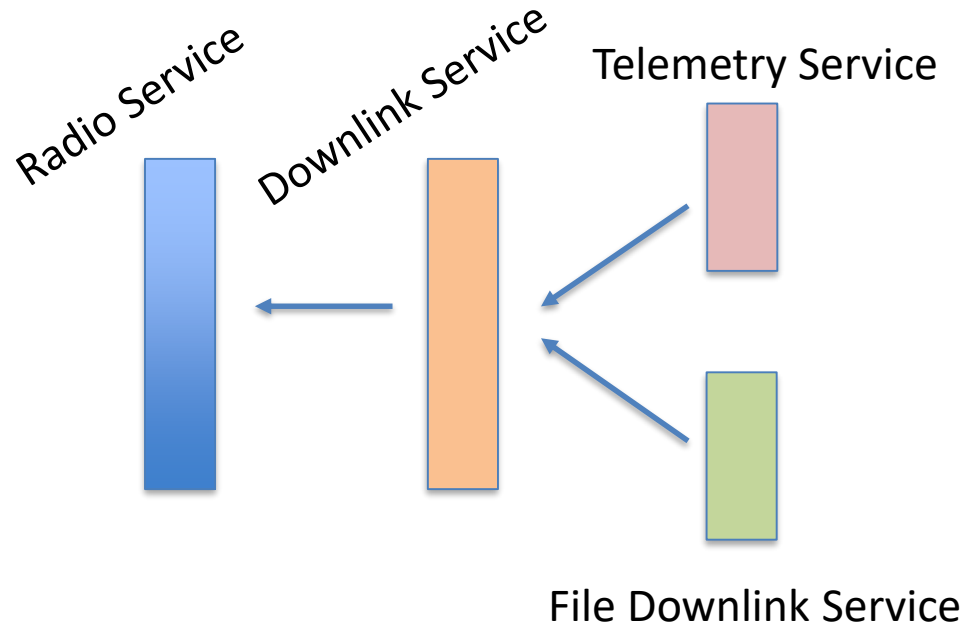
How is data passed from one task to another?

Uplink



How is data passed from one task to another?

Downlink



- Telemetry and file downlink packets are sent to the downlink task for filtering and packetization
- Downlink task send packetized data to the radio task for downlink.
- Data volume flow control must be implemented

How is data passed from one task to another?

FSW Health Service

Intended to check that FSW threads are responsive

1. no thread has been halted
2. no thread is in an infinite loop (stuck in the `process()` call)
3. no thread is stuck waiting for a reply from another thread
4. critical processing is initiated and makes progress as expected

Any failures result in system reboot

Health in MSL FSW

The MSL HEALTH module periodically does the following:

- all threads are still present and running
- send a 'ping' message to each task and wait for a reply
- if a reply to a ping is not received
 - after T_1 seconds, generate a 'WARNING' message
 - after T_2 seconds, generate a system exception that results in a reboot

How do we ensure that the HEALTH thread is running?

HEALTH has to write certain bit patterns to a specific HW register at 8Hz
if the hardware detects that bit pattern was not been written, it reboots

Agenda

- Flight Software and Avionics Systems (Section 348) Overview
- Intro to Flight Software (FSW)
- Flight Software Services
 - Commanding
 - Telemetry
 - Uplink
 - Downlink
 - Health
- Execution Platforms
 - Hardware
 - No Operating System / Bare Metal
 - Real-Time Operating System (RTOS)
- Threads
 - Concurrent Programming
 - States
 - Scheduling
- Scheduling Schemes
 - Preemptive scheduling
 - Priority Preemptive scheduling
- Thread Synchronization
 - Interference
 - Atomic Operations
 - Race Conditions
 - Shared Resources
 - Mutual Exclusion
 - Interrupt Locking
 - Semaphores
- Mars Pathfinder Priority Inversion Problem
 - What Happened
 - Priority Inversion Explained
 - Handling Priority Inversion
 - Programming Considerations with Semaphores
- The Producer-Consumer Problem
 - Scenario/Limitations Explained
 - Shared Bounded Queue
 - Using 1 semaphore
 - Using 3 semaphores
- Message Passing
 - Thread communication
 - Example application in a state machine
 - Issues in Message Passing
 - Pure Message Passing
 - Sharing a Resource with Message Passing
- The Sol 200 Anomaly

Execution Platforms

- Embedded Microcontrollers/Microprocessors
- Radiation tolerant hardware
- Memory, Performance, I/O bound



Execution Platforms

- No operating system
 - Cyclic executive
 - Can perform all tasks in a cycle before deadline
 - Simple I/O interfaces
 - Single-threaded

Execution Platforms

- Operating system (RTOS)
 - Scheduling of services
 - Priority preemptive scheduling
 - Complex I/O interfaces
 - Multi-threaded

Agenda

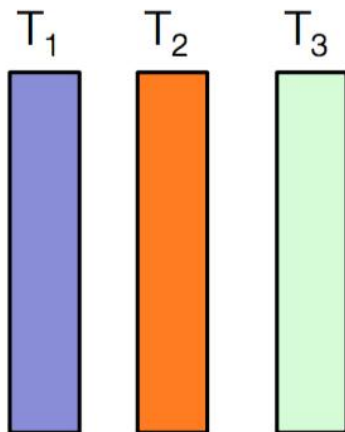
- Flight Software and Avionics Systems (Section 348) Overview
- Intro to Flight Software (FSW)
- Flight Software Services
 - Commanding
 - Telemetry
 - Uplink
 - Downlink
 - Health
- Execution Platforms
 - Hardware
 - No Operating System / Bare Metal
 - Real-Time Operating System (RTOS)
- **Threads**
 - Concurrent Programming
 - States
 - Scheduling
- Scheduling Schemes
 - Preemptive scheduling
 - Priority Preemptive scheduling
- Thread Synchronization
 - Interference
 - Atomic Operations
 - Race Conditions
 - Shared Resources
 - Mutual Exclusion
 - Interrupt Locking
 - Semaphores
- Mars Pathfinder Priority Inversion Problem
 - What Happened
 - Priority Inversion Explained
 - Handling Priority Inversion
 - Programming Considerations with Semaphores
- The Producer-Consumer Problem
 - Scenario/Limitations Explained
 - Shared Bounded Queue
 - Using 1 semaphore
 - Using 3 semaphores
- Message Passing
 - Thread communication
 - Example application in a state machine
 - Issues in Message Passing
 - Pure Message Passing
 - Sharing a Resource with Message Passing
- The Sol 200 Anomaly

Concurrent Programming with Threads

We consider a programming model in which there are multiple threads that execute asynchronously with each other.

Each thread executes a (traditional) sequential program.

Threads may interact with each other by accessing shared memory.



Threads when
run in isolation



Threads running concurrently
on a **single CPU**

Note: sometimes threads
are called **tasks**
(e.g., in VxWorks)

Note: we will not
consider multi-core
processors here

Thread State

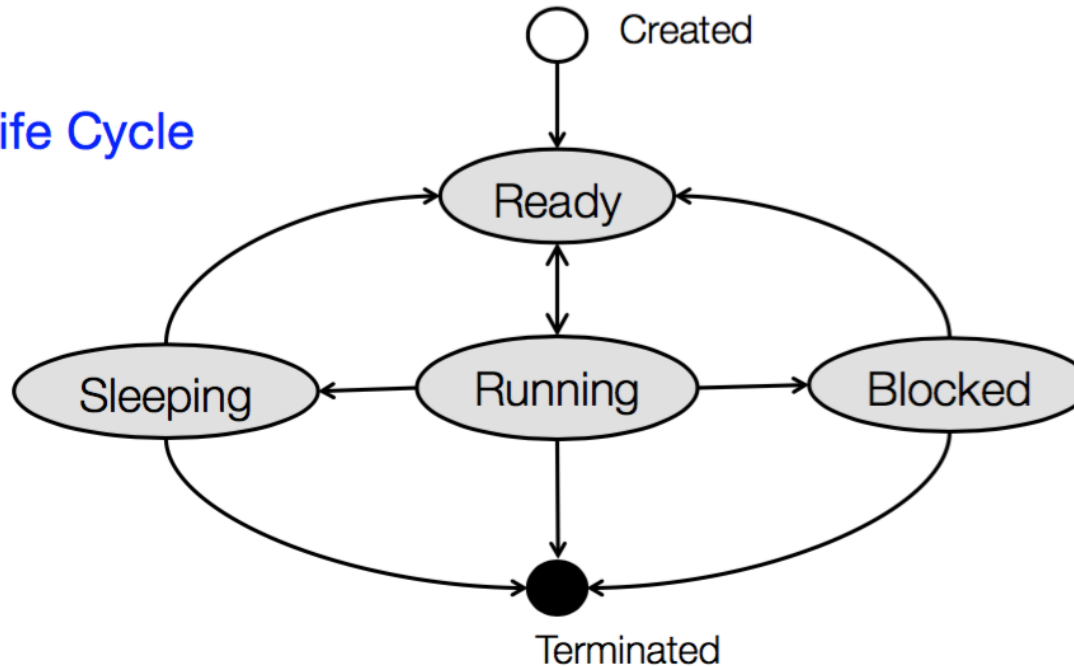
Each thread is associated with

private thread-local state -- for local data only the thread can access
(e.g., the function call stack, file table, registers)

global state -- shared with other threads

including the
program counter

Thread Life Cycle



Thread Scheduling

Golden Rule: keep the CPU busy doing **useful** work

Examples of what is not useful

- waiting on I/O from a hardware device
- waiting for some condition to become true (queue to become nonempty)

Two types of scheduling algorithms

Cooperative Scheduling

Each thread decides when to give up the CPU, by executing a special **yield** operation

Preemptive Scheduling

Threads have no control over when they are scheduled – the operating system decides

Agenda

- Flight Software and Avionics Systems (Section 348) Overview
- Intro to Flight Software (FSW)
- Flight Software Services
 - Commanding
 - Telemetry
 - Uplink
 - Downlink
 - Health
- Execution Platforms
 - Hardware
 - No Operating System / Bare Metal
 - Real-Time Operating System (RTOS)
- Threads
 - Concurrent Programming
 - States
 - Scheduling
- Scheduling Schemes
 - Preemptive scheduling
 - Priority Preemptive scheduling
- Thread Synchronization
 - Interference
 - Atomic Operations
 - Race Conditions
 - Shared Resources
 - Mutual Exclusion
 - Interrupt Locking
 - Semaphores
- Mars Pathfinder Priority Inversion Problem
 - What Happened
 - Priority Inversion Explained
 - Handling Priority Inversion
 - Programming Considerations with Semaphores
- The Producer-Consumer Problem
 - Scenario/Limitations Explained
 - Shared Bounded Queue
 - Using 1 semaphore
 - Using 3 semaphores
- Message Passing
 - Thread communication
 - Example application in a state machine
 - Issues in Message Passing
 - Pure Message Passing
 - Sharing a Resource with Message Passing
- The Sol 200 Anomaly

Preemptive Scheduling

Requires special hardware support

- support for *interrupts*

Whenever a clock interrupt occurs

- the operating system scheduler takes over, and decides which thread should run next
- period of time between interrupts is called the *time slice*

What impact does the setting of the time slice have on the system?

On each interrupt, the operating system has to

- save the state of the current thread
- evaluate which threads are ready to run
- schedule the thread that *should* be run next

Setting the time slice

too low → greater scheduling overhead

too high → higher latency between thread switching

Priority Preemptive Scheduling

Most Real-Time Operating Systems support *Priority-Preemptive Scheduling*:

Each thread has an assigned priority (often 0..255)

Scheduler gives CPU to the highest priority thread that is *Ready*

Note that a thread may be *preempted* by

- another **higher** priority thread
- an interrupt service routine (ISR)

Note: in some operating systems (VxWorks) priority 255 is the lowest priority; in others (Green Hills), it is the highest

An ISR is executed when an interrupt happens

In general, an ISR may preempt another ISR

If you need to ensure that a certain block of code is executed **atomically**, you can disable (**mask**) all interrupts. Typically, ISRs mask interrupts while they are executing.

Priority Preemptive Scheduling

Q. What are the pros/cons of preemptive priority scheduling?

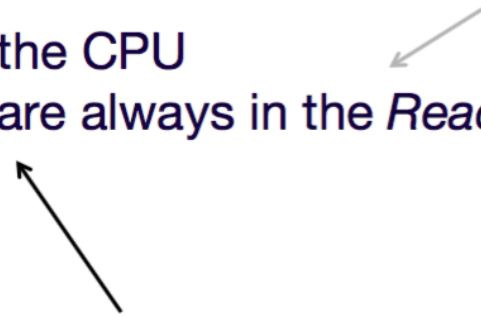
Pros

- can allow “important” threads to run with low latency whenever ready
- more predictable system behavior

Cons

- low priority thread may never get the CPU
- what if you have 2 threads which are always in the *Ready* state?

so-called CPU-bound threads
e.g., image compression
memory scrubbing

Two arrows originate from the text box. One arrow points to the word "CPU" in the first bullet point of the "Cons" section. The other arrow points to the phrase "always in the Ready state?" in the second bullet point.

Some schedulers can (optionally) perform round-robin scheduling among *Ready* threads at the same priority – effectively sharing the CPU within a priority level. (VxWorks has this feature, though MSL did not use it.)

Agenda

- Flight Software and Avionics Systems (Section 348) Overview
- Intro to Flight Software (FSW)
- Flight Software Services
 - Commanding
 - Telemetry
 - Uplink
 - Downlink
 - Health
- Execution Platforms
 - Hardware
 - No Operating System / Bare Metal
 - Real-Time Operating System (RTOS)
- Threads
 - Concurrent Programming
 - States
 - Scheduling
- Scheduling Schemes
 - Preemptive scheduling
 - Priority Preemptive scheduling
- Thread Synchronization
 - Interference
 - Atomic Operations
 - Race Conditions
 - Shared Resources
 - Mutual Exclusion
 - Interrupt Locking
 - Semaphores
- Mars Pathfinder Priority Inversion Problem
 - What Happened
 - Priority Inversion Explained
 - Handling Priority Inversion
 - Programming Considerations with Semaphores
- The Producer-Consumer Problem
 - Scenario/Limitations Explained
 - Shared Bounded Queue
 - Using 1 semaphore
 - Using 3 semaphores
- Message Passing
 - Thread communication
 - Example application in a state machine
 - Issues in Message Passing
 - Pure Message Passing
 - Sharing a Resource with Message Passing
- The Sol 200 Anomaly

Thread Interference

```
int x = 0 ;
```

T_1

```
x = x + 1 ;
```

T_2

```
x = x + 2 ;
```

Q. What are the possible outcomes if we run T_1 and T_2 concurrently with a round-robin scheduler?

A. x may be 3, 1 or 2. (Why?)

Q. Does your answer change if we use a priority-preemptive scheduler and T_1 and T_2 have different priorities?

Atomic Operations

An operation is said to be **atomic** if its execution cannot be interrupted.

From the point of view of other threads, the only visible state is *either the state before the operation started or after the operation has fully completed*

Primitive atomic operations available on modern CPUs
read / write of a **single word** (4 bytes on a 32-bit processor)
test-and-set
compare-and-swap

Using primitive operations, one can build a library of richer **synchronization primitives** to make concurrent programs easier to write.

More on Atomic Operations

Note that atomicity is not always apparent when looking at source code

```
x++ ;
```

looks like a single operation but is not atomic
compiles into 3 instructions { read(x) ; add 1 ; write(x) }

Do the following threads behave as expected?

shared variable

```
int x = 0 ;
```

T_1

```
extern int buf[N] ;  
while (x == 0) {  
    sleep(1) ;  
}  
// read values from buf
```

T_2

```
extern int buf[N] ;  
// write values into buf  
x++ ;
```

sleep(k) causes thread to sleep for k seconds


Yes – T_1 will only start reading values from buf after T_2 has finished writing

Race Conditions

A **data race** occurs when two threads access the same shared data, with no mutual coordination, and at least one of the threads is performing **writes**.

T_1

```
if (x < 5) {  
    y = x+1 ;  
    ASSERT(y < 6) ;  
}
```



T_2

```
x = 251 ;
```

In what situations can this assertion fail?

One way to prevent race conditions is to enforce *mutual exclusion*

Shared Resources

A **shared resource** is a hardware device or a data structure (in memory) that may be accessed by multiple threads.

In many cases, the resource

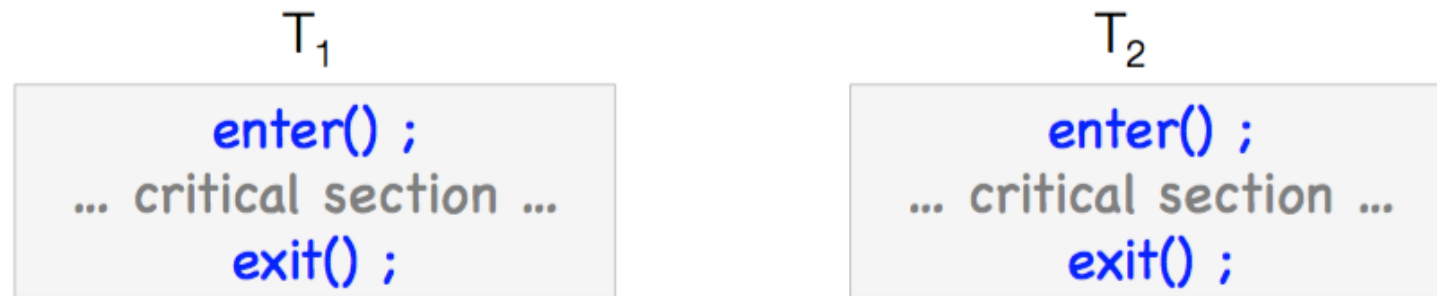
- cannot be copied
- must be written by at least one thread

In such cases, we need a protocol so that a thread may access the resource **exclusively** for a while without interference from other threads using that shared resource.

(Note: okay to run threads that not using the resource, since they are not interfering.)

Mutual Exclusion

A protocol for ensuring that at most one thread can execute a piece of code (which is called a *critical section* of code)



The `enter()` code allows at most one thread to gain entry to the critical section. The `exit()` code does cleanup after the thread is done so another thread can perform its critical section.

Any other threads that try to enter the critical section are **blocked** until the current thread is done.

Interrupt locking

Most real-time/embedded operating systems provide special primitives that allows **all interrupts to be disabled or enabled**.

Q. Can you think of how you may use this feature to implement mutual exclusion?

```
lockInterrupts() ; // disable all interrupts  
// critical section  
unlockInterrupts() ; // enable all interrupts
```

Q. Is this a desirable way to implement mutual exclusion?
What are the pros and cons?

Semaphores (aka “Locks”)

Invented ~ 1962 by Edsger W. Dijkstra



Two operations:

<code>SemTake()</code>	--- acquires the semaphore if it is <i>available</i>
<code>SemGive()</code>	--- makes the semaphore <i>available</i>

Implementing mutual exclusion with semaphores is trivial

```
SemTake() ;  
// critical section  
SemGive() ;
```

The key property of semaphores is that whenever a thread attempts a `SemTake()` operation that fails, the thread becomes *Blocked* and will not be scheduled to run. When a `SemGive()` happens, the operating system will give the semaphore to one of the threads waiting for it, and that thread will become *Running*.

Agenda

- Flight Software and Avionics Systems (Section 348) Overview
- Intro to Flight Software (FSW)
- Flight Software Services
 - Commanding
 - Telemetry
 - Uplink
 - Downlink
 - Health
- Execution Platforms
 - Hardware
 - No Operating System / Bare Metal
 - Real-Time Operating System (RTOS)
- Threads
 - Concurrent Programming
 - States
 - Scheduling
- Scheduling Schemes
 - Preemptive scheduling
 - Priority Preemptive scheduling
- Thread Synchronization
 - Interference
 - Atomic Operations
 - Race Conditions
 - Shared Resources
 - Mutual Exclusion
 - Interrupt Locking
 - Semaphores
- Mars Pathfinder Priority Inversion Problem
 - What Happened
 - Priority Inversion Explained
 - Handling Priority Inversion
 - Programming Considerations with Semaphores
- The Producer-Consumer Problem
 - Scenario/Limitations Explained
 - Shared Bounded Queue
 - Using 1 semaphore
 - Using 3 semaphores
- Message Passing
 - Thread communication
 - Example application in a state machine
 - Issues in Message Passing
 - Pure Message Passing
 - Sharing a Resource with Message Passing
- The Sol 200 Anomaly

The Mars Pathfinder priority inversion problem

Devices connected to 1553 bus

Flight software responsible for scheduling bus transactions
must schedule at fixed rate (8 Hz)

Two threads in software:

- `bc_sched` -- sets up transactions on bus for next cycle (highest priority)
- `bc_dist` -- collects data for current cycle (2nd highest priority)

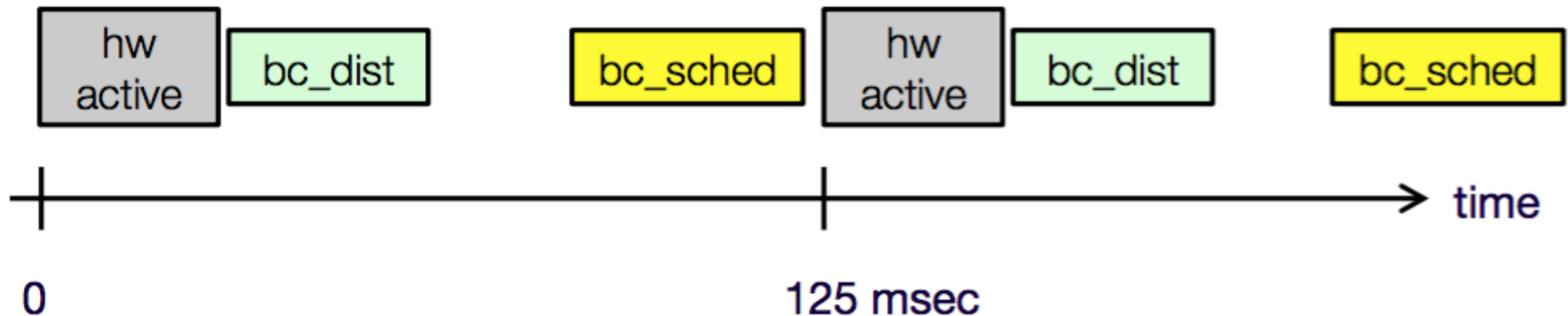


Figure not to scale

The Mars Pathfinder priority inversion problem

Failure after landing:

bc_dist task did not finish before bc_sched was due to start
→ response was to reset the computer

What was happening:

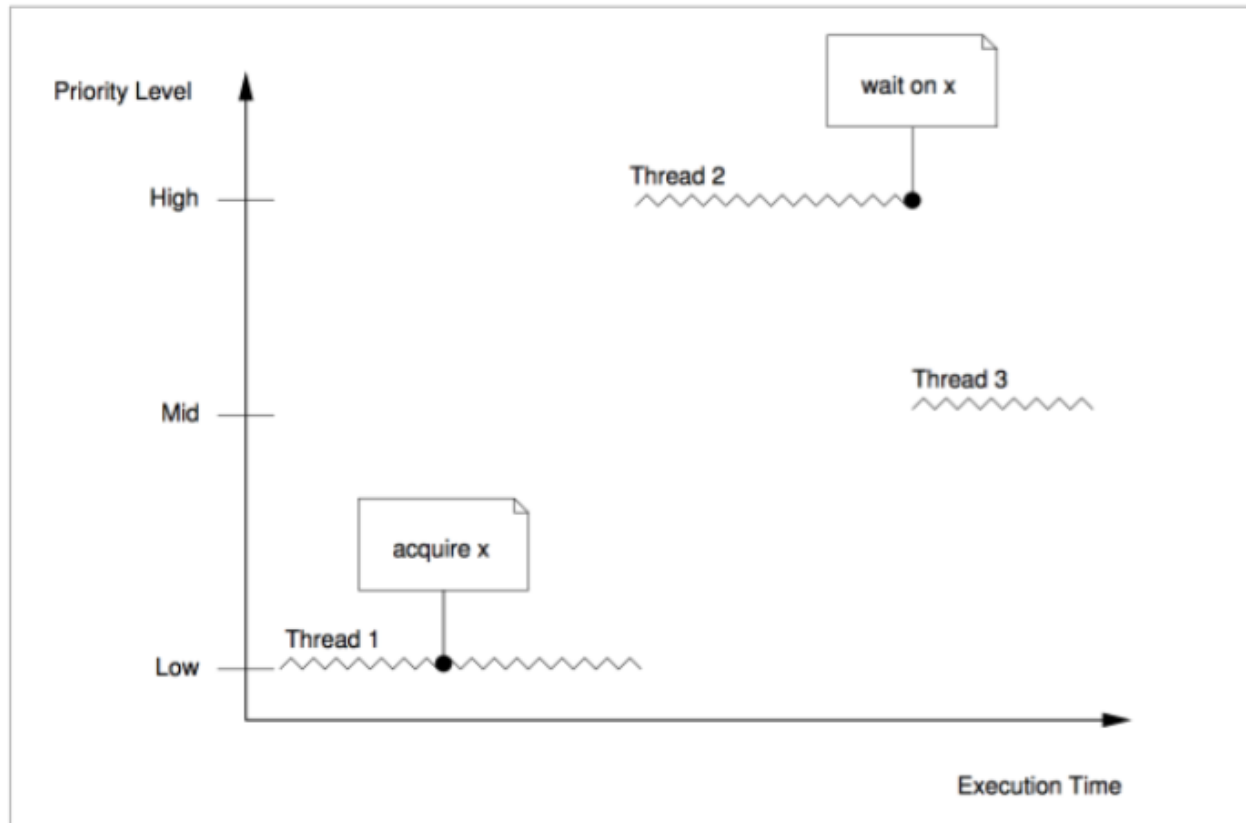
The bc_dist task was blocked on a semaphore that was held by a low priority ASI/MET thread that managed an instrument for meteorological science.

The ASI/MET thread had been preempted by other, medium priority, threads and continued to hold on to the semaphore.

Fix was to enable the priority inheritance flag on the semaphore

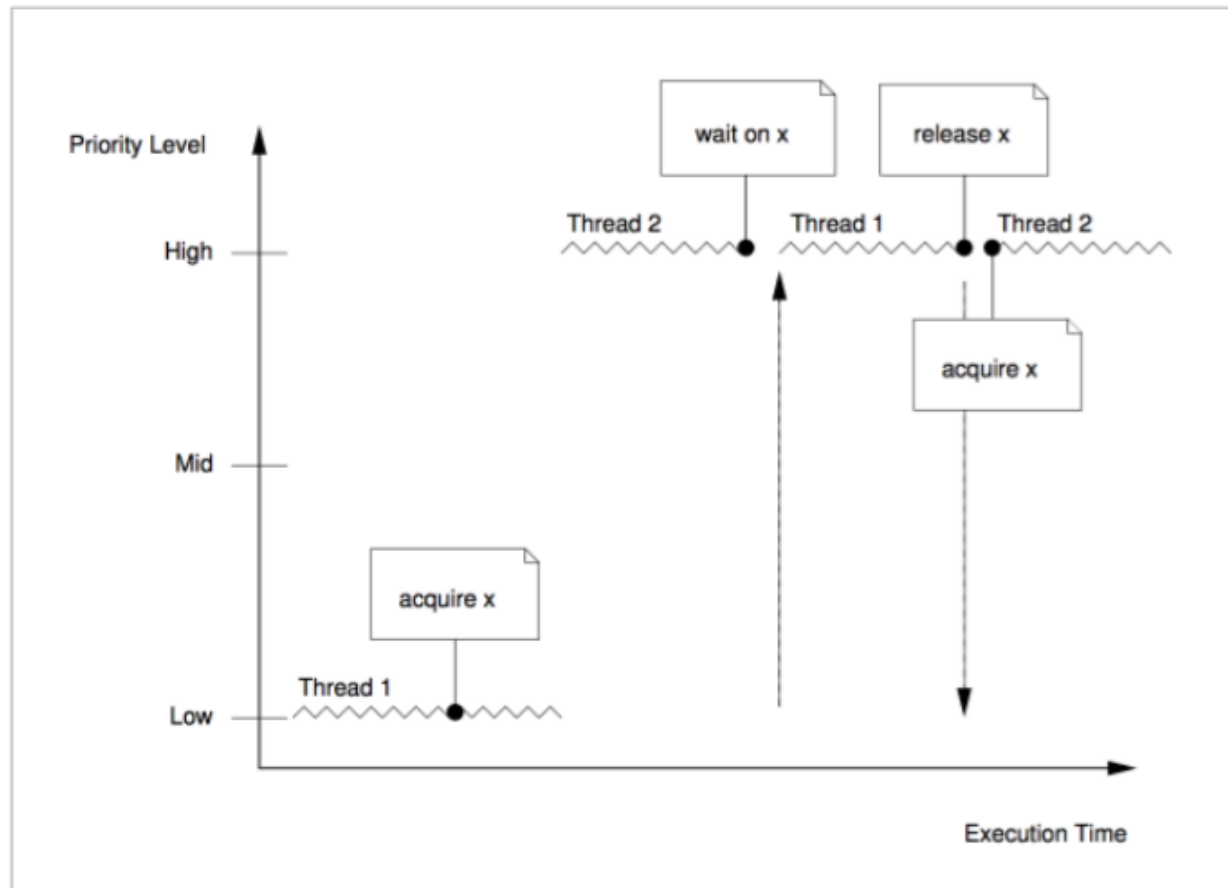
Google *[“What really happened on Mars”](#)* by Mike Jones (also see followup by Glenn Reeves)

Priority Inversion



Made famous by [Mars Pathfinder](#)

Handling Priority Inversion



This behavior is enabled by using mutexes

Accessing multiple resources

Suppose we have N shared resources in the system.

We could protect them all with a single semaphore, but then if two threads want to use two different resources, they have to unnecessarily wait for each other.

So, we protect each resource with its own semaphore $S_1 \dots S_N$

What could go wrong?

Suppose thread T_1 wants to use resources 2, 5 and 7 and thread T_2 wants to use 2, 4 and 5.

What if T_1 executes: $\text{Lock}(S_2) ; \text{Lock}(S_5) ; \text{Lock}(S_7) ;$
and T_2 executes: $\text{Lock}(S_5) ; \text{Lock}(S_4) ; \text{Lock}(S_2) ;$

This could cause **deadlock**, with the threads permanently blocked:

T_1 waiting for S_5 and
 T_2 waiting for S_2

A simple way to avoid this problem is to require either
(a) no thread can grab more than 1 resource at a time, or
(b) all resources must be acquired in the same (global) order

Programming Considerations with Semaphores

Semaphores should be used with care in a real-time application

- they are low-level primitives, should not typically be used directly (instead, use high-level application-specific functions that hide semaphores from most clients)
- blocking behavior means critical thread could block for a noncritical thread (priority inversion safe semaphores are a band-aid, not a solution)

Good practices

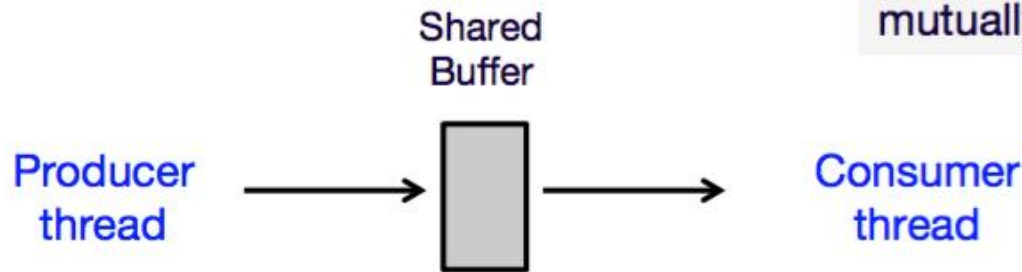
- Limit time spent inside critical section – e.g., don't call outside functions because they may take too long, or block on a semaphore themselves (Note: developer maintaining that function may change behavior in the future)
- Ensure multiple semaphores are acquired in the same order by all threads

Agenda

- Flight Software and Avionics Systems (Section 348) Overview
- Intro to Flight Software (FSW)
- Flight Software Services
 - Commanding
 - Telemetry
 - Uplink
 - Downlink
 - Health
- Execution Platforms
 - Hardware
 - No Operating System / Bare Metal
 - Real-Time Operating System (RTOS)
- Threads
 - Concurrent Programming
 - States
 - Scheduling
- Scheduling Schemes
 - Preemptive scheduling
 - Priority Preemptive scheduling
- Thread Synchronization
 - Interference
 - Atomic Operations
 - Race Conditions
 - Shared Resources
 - Mutual Exclusion
 - Interrupt Locking
 - Semaphores
- Mars Pathfinder Priority Inversion Problem
 - What Happened
 - Priority Inversion Explained
 - Handling Priority Inversion
 - Programming Considerations with Semaphores
- The Producer-Consumer Problem
 - Scenario/Limitations Explained
 - Shared Bounded Queue
 - Using 1 semaphore
 - Using 3 semaphores
- Message Passing
 - Thread communication
 - Example application in a state machine
 - Issues in Message Passing
 - Pure Message Passing
 - Sharing a Resource with Message Passing
- The Sol 200 Anomaly

The Producer-Consumer problem

Suppose we have two threads in a system
the Producer thread generates data periodically
the Consumer periodically consumes the data



Using semaphores, we can ensure mutually exclusive access to the buffer

What is the limitation of this design?

Producer must wait while the Consumer is reading, and vice versa

Shared bounded queue, using 1 semaphore

`enqueue(V)`

add value V to the front

`V = dequeue()`

remove value from the back



Using 1 semaphore

`Mutex Qsem ;`

```
void enqueue(int V) {  
    SemTake(Qsem) ;  
    // add V to array  
    SemGive(Qsem) ;  
}
```

```
int dequeue() {  
    SemTake(Qsem) ;  
    // remove V from array  
    SemGive(Qsem) ;  
    return V ;  
}
```

What are the pros/cons of this implementation?

How do you handle the buffer empty/full conditions?

Shared bounded queue, using 3 semaphores

`enqueue(V)`

add value V to the front

`V = dequeue()`

remove value from the back



Using 3 semaphores

```
CountingSem  NotEmpty ; // init 0
CountingSem  NotFull ;   // init N
Mutex        Qsem ;
```

```
void enqueue(int V) {
    SemTake(NotFull) ;
    SemTake(Qsem) ;
    // add V to array
    SemGive(Qsem) ;
    SemGive(NotEmpty) ;
}
```

```
int dequeue() {
    SemTake(NotEmpty) ;
    SemTake(Qsem) ;
    // remove V from array
    SemGive(Qsem) ;
    SemGive(NotFull) ;
    return V ;
}
```

How does this behave when the buffer is empty/full?

It causes the calling thread to block

Agenda

- Flight Software and Avionics Systems (Section 348) Overview
- Intro to Flight Software (FSW)
- Flight Software Services
 - Commanding
 - Telemetry
 - Uplink
 - Downlink
 - Health
- Execution Platforms
 - Hardware
 - No Operating System / Bare Metal
 - Real-Time Operating System (RTOS)
- Threads
 - Concurrent Programming
 - States
 - Scheduling
- Scheduling Schemes
 - Preemptive scheduling
 - Priority Preemptive scheduling
- Thread Synchronization
 - Interference
 - Atomic Operations
 - Race Conditions
 - Shared Resources
 - Mutual Exclusion
 - Interrupt Locking
 - Semaphores
- Mars Pathfinder Priority Inversion Problem
 - What Happened
 - Priority Inversion Explained
 - Handling Priority Inversion
 - Programming Considerations with Semaphores
- The Producer-Consumer Problem
 - Scenario/Limitations Explained
 - Shared Bounded Queue
 - Using 1 semaphore
 - Using 3 semaphores
- Message Passing
 - Thread communication
 - Example application in a state machine
 - Issues in Message Passing
 - Pure Message Passing
 - Sharing a Resource with Message Passing
- The Sol 200 Anomaly

Thread Communication using Message Passing

A common pattern for exchanging data between threads is to send **messages**.

A sender (the producer) calls

Send(M, R) to send a message with data **M** to the queue named **R**

The receiver (consumer) calls

M = Recv(R) to receive a message with data **M** from the queue named **R**

Typically, many threads could send to the same message queue – so the underlying implementation must ensure that **Send/Recv** operations are atomic.

Message Passing Example:

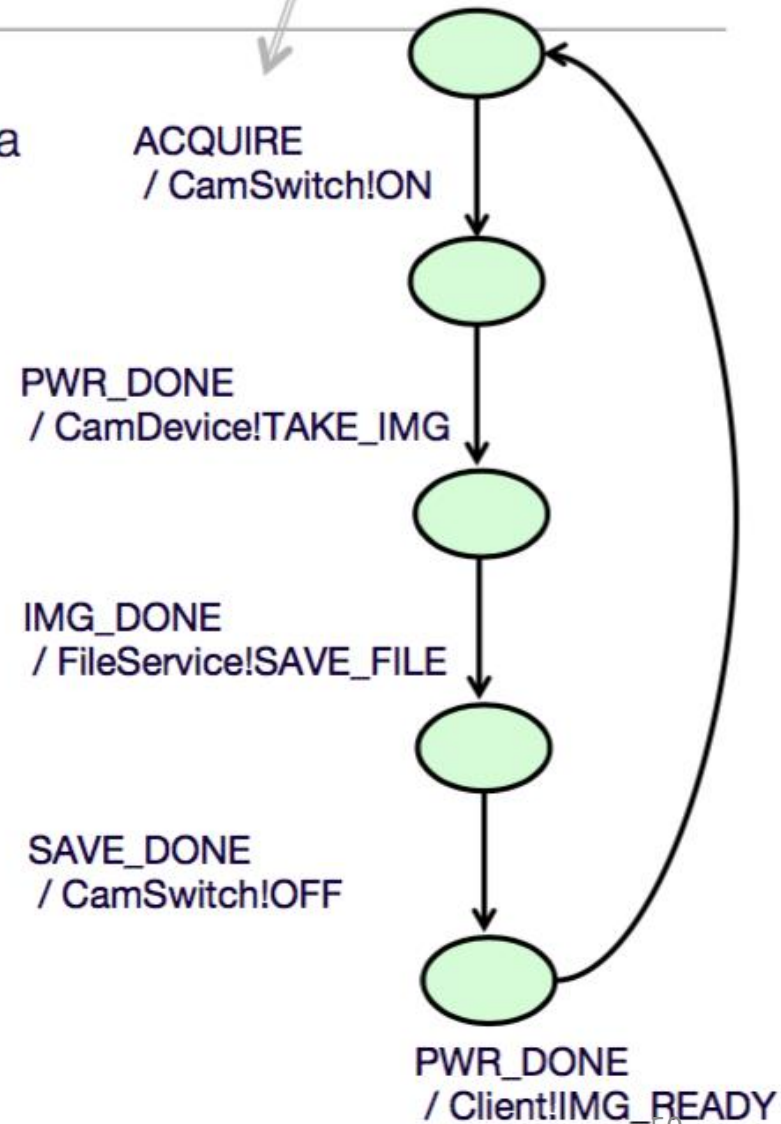
A simple Imaging State Machine

Implement a thread for taking images with a camera

```
ThreadId Client = -1 ;
```

```
while (true) {  
    M = Recv() ;  
    if (M == ACQUIRE) {  
        ClientId = M.Client  
        Send(ON, CamSwitch)  
    } else if (M == PWR_DONE) {  
        Send(TAKE_IMG, CamDevice)  
    } else if (M == IMG_DONE) {  
        Send(SAVE_FILE, FileService)  
    } else if (M == SAVE_DONE) {  
        Send(OFF, CamSwitch)  
    } else if (M == PWR_DONE) {  
        Send(IMG_READY, Client)  
    } else { ... error ... }  
}
```

notation: on receiving ACQUIRE message, send ON message to CamSwitch thread



Issues in Message Passing

What are the pros/cons of using our previous shared queue implementation (using 3 semaphores) for doing inter-thread messaging?

The use of semaphores means that ISR calls cannot send messages

Blocking a producer when full is a bad choice if the producer is a critical task

Pure Message Passing and Shared Memory

In a **pure message passing** system, the threads do not access any common shared memory directly. If a producer generates data, it sends the data item in a message.

This has the nice property that no thread synchronization is required – when the consumer operates on data, it has a local copy that cannot be modified by any other threads.

In practice, however, pure message passing is impractical if threads need to share large data items (such as images), because

- it requires extra storage (for making a copy for the consumer)
- it requires extra time (for making the copy)

We could fall back to using a single shared buffer protected by a semaphore

But that's also not a desirable solution

-- if either the producer or consumer is a critical thread, we don't want it to block

Sharing a resource with message passing


One approach to sharing memory with message passing is to use a notion of **ownership**

At any moment, the shared buffer is owned by a thread (say, either the Producer or Consumer). Ownership is transferred from one thread to another using messages (note that these messages are very small).

When the Producer has finished writing the shared memory, it assigns ownership to the Consumer and sends it a message.

When the Consumer receives the message, it can access the memory without using a semaphore (i.e., without risk of blocking). When it is done, it gives ownership back to the Producer by sending it a message.

To catch coding errors, before a thread starts using the resource, it could perform **ASSERT(Owner == CurrentThreadId())**

 Global variable indicating current owner

Agenda

- Flight Software and Avionics Systems (Section 348) Overview
- Intro to Flight Software (FSW)
- Flight Software Services
 - Commanding
 - Telemetry
 - Uplink
 - Downlink
 - Health
- Execution Platforms
 - Hardware
 - No Operating System / Bare Metal
 - Real-Time Operating System (RTOS)
- Threads
 - Concurrent Programming
 - States
 - Scheduling
- Scheduling Schemes
 - Preemptive scheduling
 - Priority Preemptive scheduling
- Thread Synchronization
 - Interference
 - Atomic Operations
 - Race Conditions
 - Shared Resources
 - Mutual Exclusion
 - Interrupt Locking
 - Semaphores
- Mars Pathfinder Priority Inversion Problem
 - What Happened
 - Priority Inversion Explained
 - Handling Priority Inversion
 - Programming Considerations with Semaphores
- The Producer-Consumer Problem
 - Scenario/Limitations Explained
 - Shared Bounded Queue
 - Using 1 semaphore
 - Using 3 semaphores
- Message Passing
 - Thread communication
 - Example application in a state machine
 - Issues in Message Passing
 - Pure Message Passing
 - Sharing a Resource with Message Passing
- The Sol 200 Anomaly

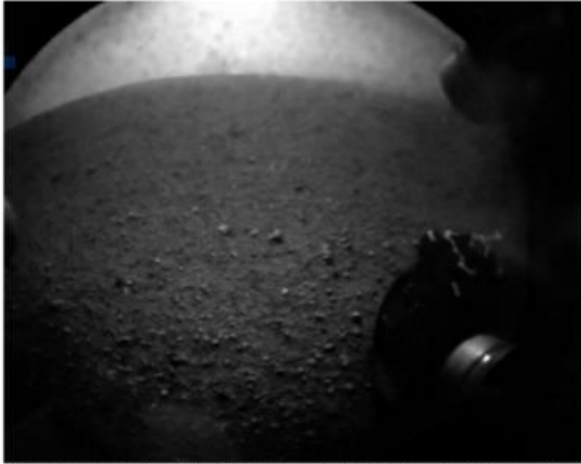
THE SOL-200 ANOMALY

16 HOURS OF TERROR

Mission Status on Sol-200 (Feb 27, 2013)

NASA's Curiosity Rover Successfully Lands on Mars

By KENNETH CHANG and JEREMY ZILAR AUGUST 5, 2012 10:55 PM 62



NASA TV, via Reuters One of the first test images from NASA's Mars Curiosity rover that helped signal that everything was operational.

Aug 5, 2012: Successful landing

EDITION: US

zdnet SEARCH

VIDEOS CIO WINDOWS 8 CLOUD INNOVATION SECURITY APPLE MORE NEWSLETTERS

MUST READ: **SERIOUS SECURITY: THREE CHANGES THAT COULD TURN THE TIDE ON HACKERS**

NASA gives Curiosity Mars rover its first major software update

An update uploaded to the mobile laboratory as it was en route from Earth to Mars was installed over the weekend, to help Curiosity carry out its experiments and not bump into things

Aug 13, 2012: Flight software upgrade for surface operations



Feb 8, 2013: First Sample Drilling

SPACE & COSMOS

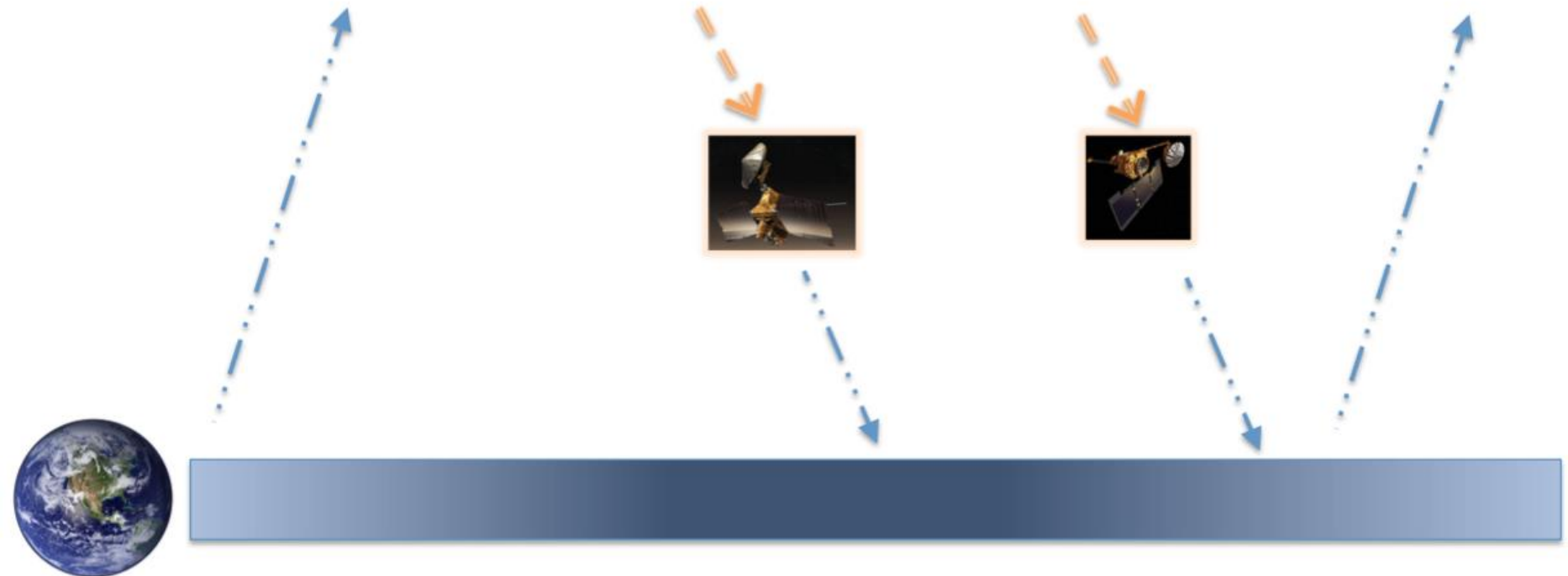
Mars Could Have Supported Life Long Ago, NASA Says

By KENNETH CHANG MARCH 12, 2013

Two images of the surface of Mars from the Opportunity rover, left, and the Curiosity rover. Scientists are studying Martian rocks for evidence of past life. NASA, via European Space Agency

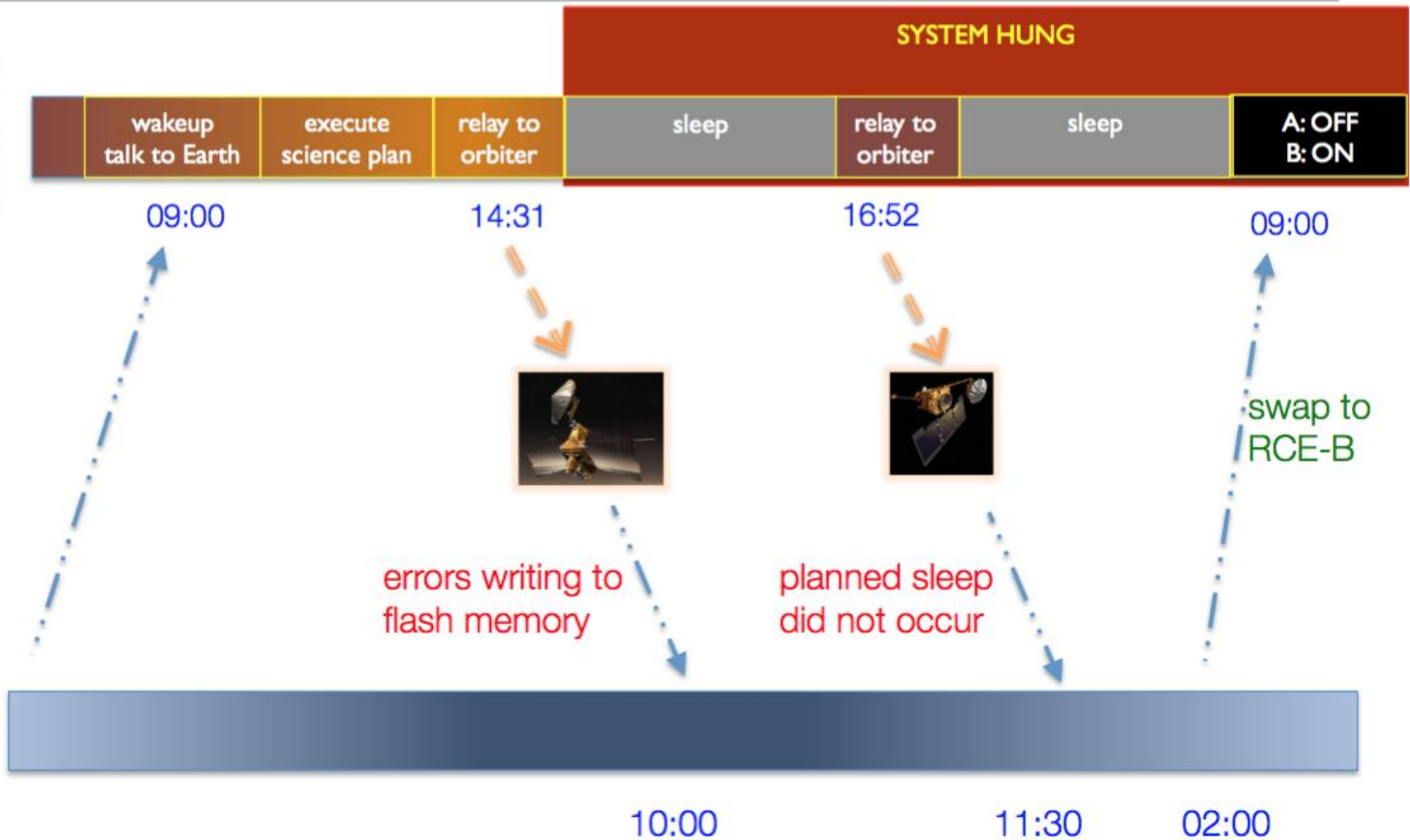
Science team on the verge of publicly announcing the major discovery that Mars was once habitable

A typical “sol” on Mars

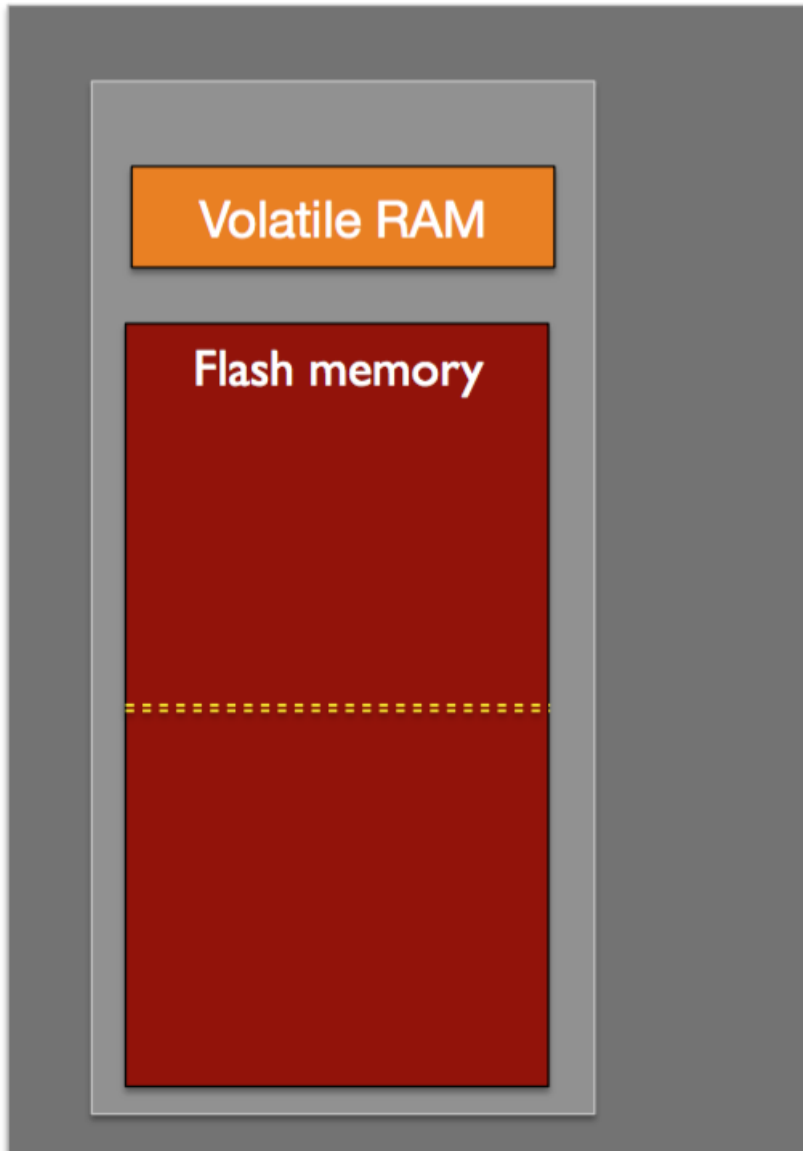


timeline not to scale

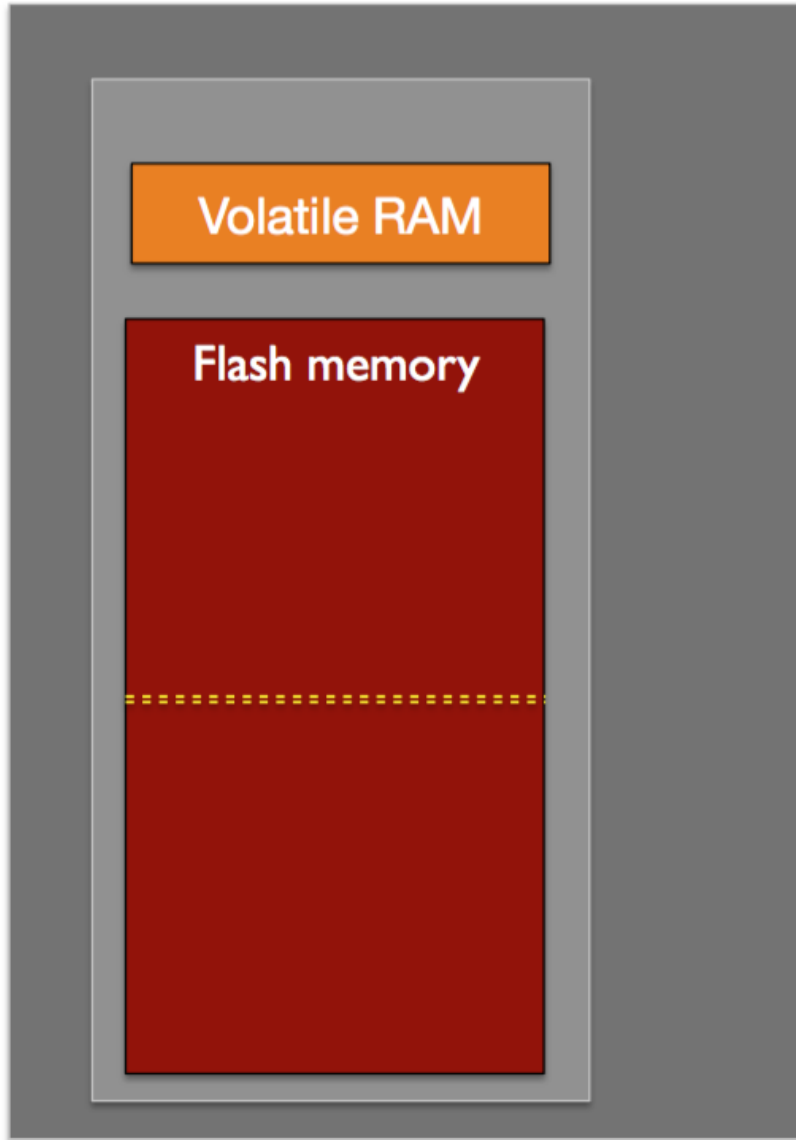
Sol-200



Data Generation

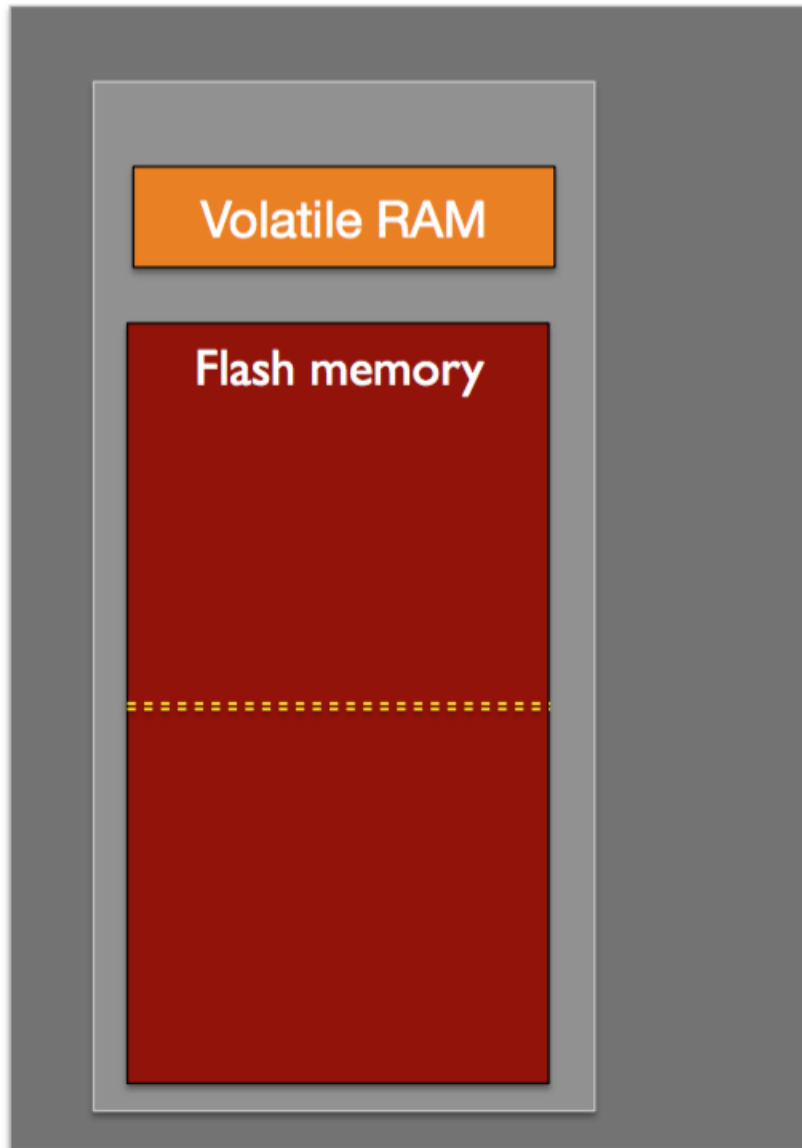


Data Generation



1. Camera acquires image

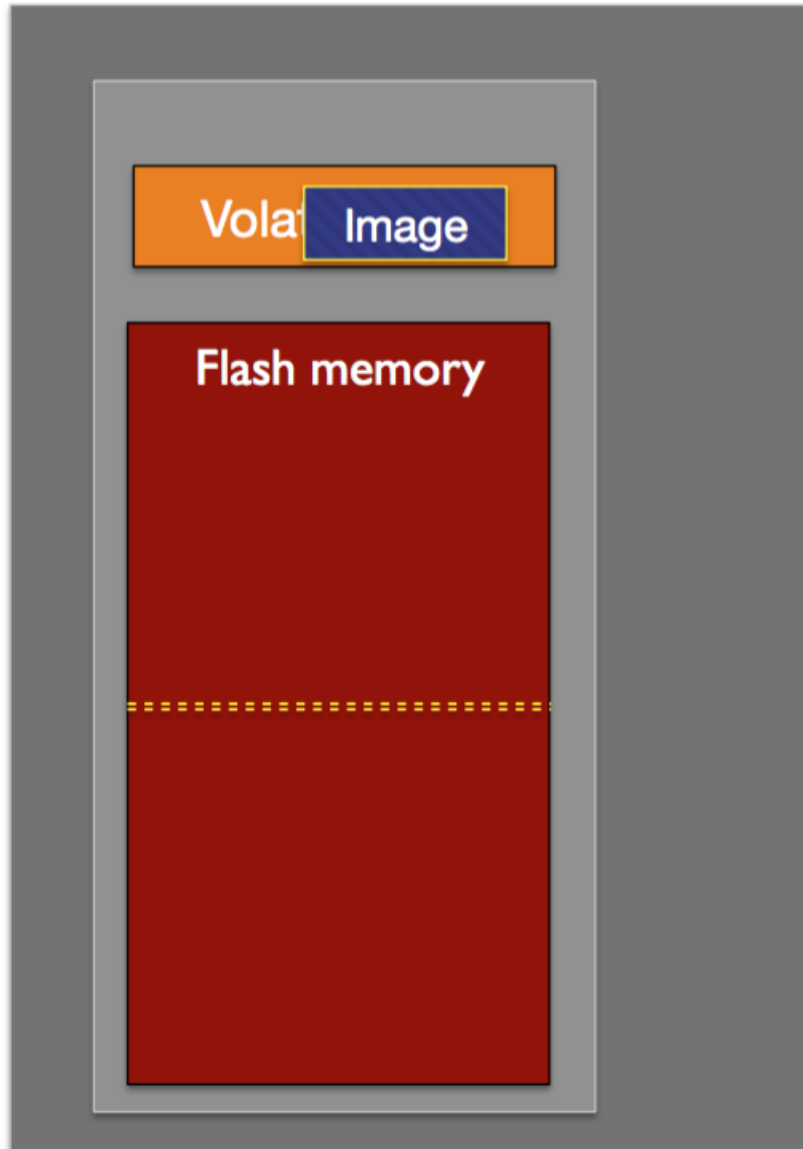
Data Generation



1. Camera acquires image

2. Requests RAM file allocation

Data Generation

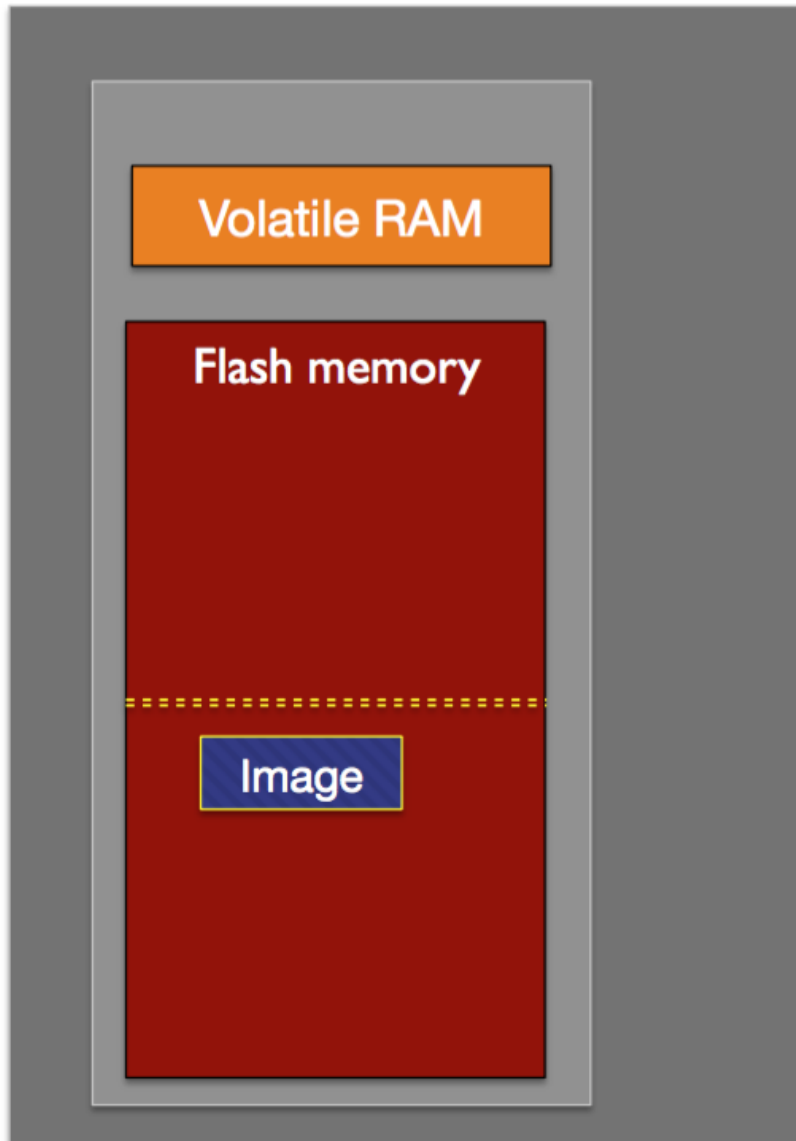


1. Camera acquires image

2. Requests RAM file allocation

3. On receiving allocation, writes image data to file

Data Generation



1. Camera acquires image


2. Requests RAM file allocation

3. On receiving allocation, writes image data to file

4. Data Manager copies file to flash
and deletes file from RAM

Data Generation when memory becomes full

what if memory is
not available?



1. Camera acquires image

2. Requests RAM file allocation

3. On receiving allocation, writes
image data to file

4. Data Manager copies file to
flash
and deletes file from RAM

Options

- client retries later
- throw away the data
- client waits for allocation

Data generation pattern

two kinds of requests

WAIT and **NOWAIT**

client waits until
memory is available
(for critical data)

client discards data if
memory is not available
(for noncritical data)

1. Camera acquires image

2. Requests RAM file allocation

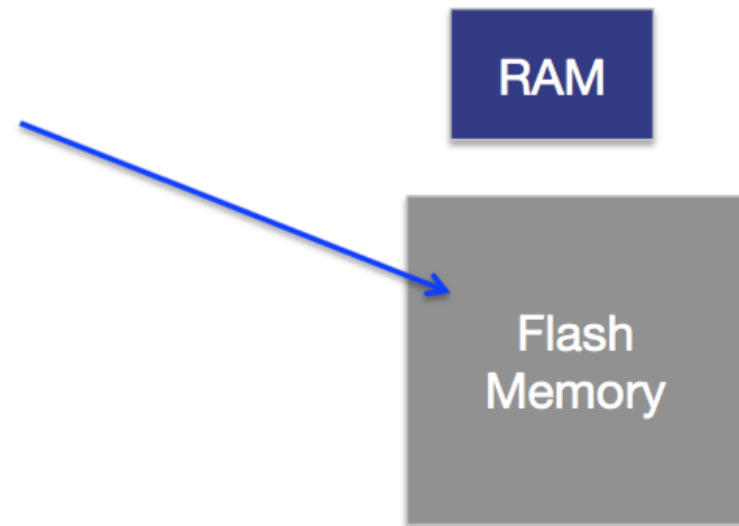
3. On receiving allocation, writes
image data to file

4. Data Manager copies file to
flash
and deletes file from RAM

Sol-200 Anomaly Reconstruction



10:00 Catastrophic failure of flash memory
→ flash filesystem becomes unwritable

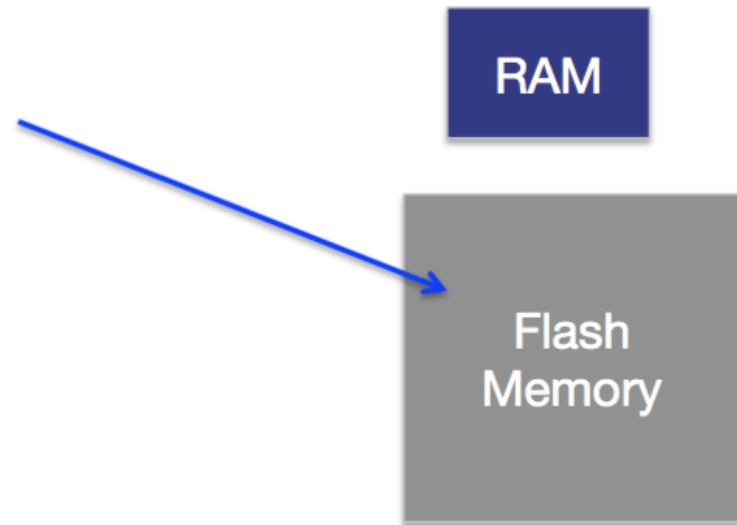


Sol-200 Anomaly Reconstruction



10:00 Catastrophic failure of flash memory
→ flash filesystem becomes unwritable

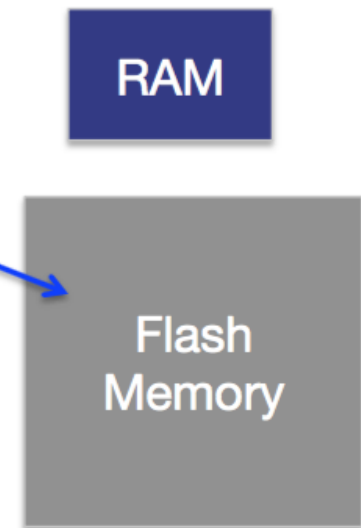
12:30 RAM fills up



Sol-200 Anomaly Reconstruction



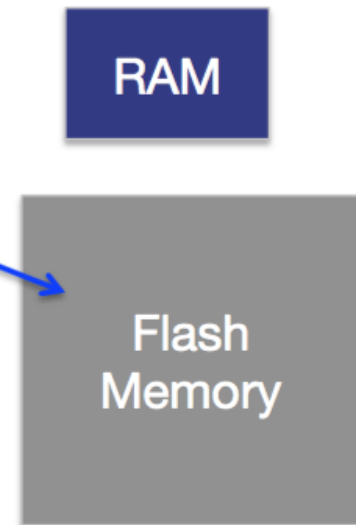
- 10:00 Catastrophic failure of flash memory
→ **flash filesystem becomes unwritable**
- 12:30 RAM fills up
- 14:30 FSM thread requests file allocation with WAIT option
→ **disables request queue**



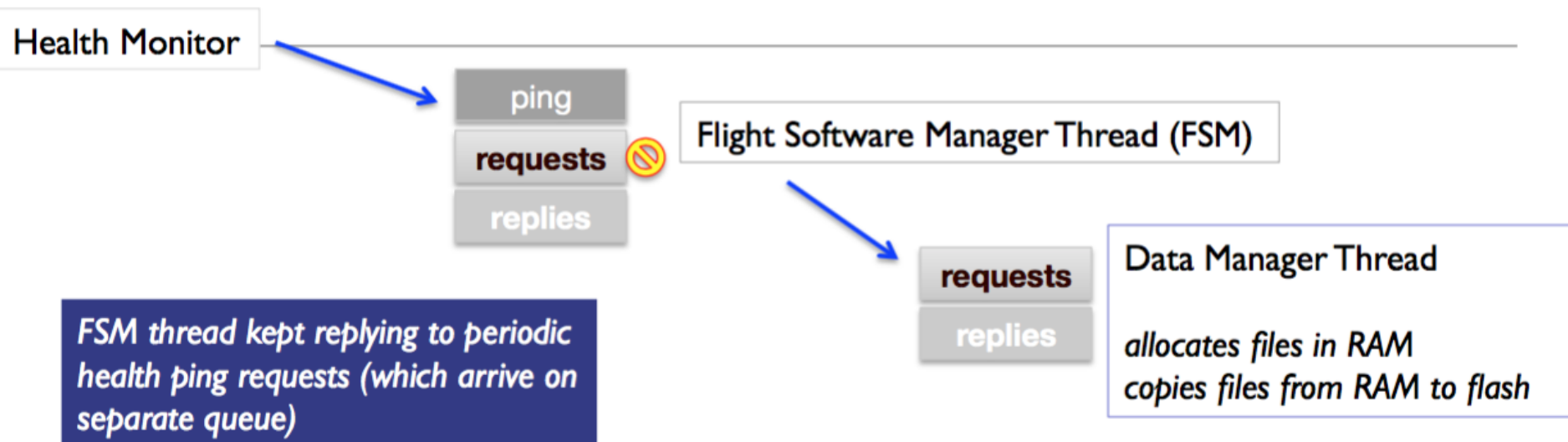
Sol-200 Anomaly Reconstruction



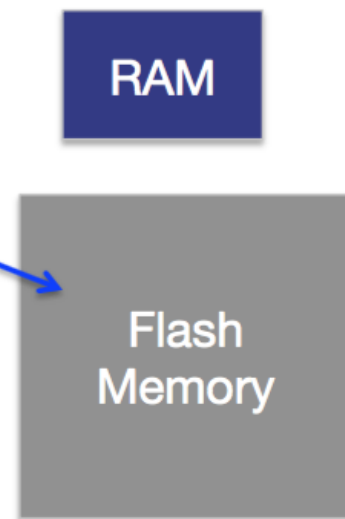
- 10:00 Catastrophic failure of flash memory
→ **flash filesystem becomes unwritable**
- 12:30 RAM fills up
- 14:30 FSM thread requests file allocation with WAIT option
→ **disables request queue**
- 15:00 Shutdown request arrives on FSW request queue
(but is not retrieved)



Sol-200 Why health monitoring failed



- 10:00 Catastrophic failure of flash memory
→ flash filesystem becomes unwritable
- 12:30 RAM fills up
- 14:30 FSM thread requests file allocation with WAIT option
→ disables request queue
- 15:00 Shutdown request arrives on FSW request queue
(but is not retrieved)



Sol-200 Recovery Steps

Run diagnostic tests to determine state of failed computer
use cross-string commanding to test flash memory
patch software to map out failed bank of memory

Patch code to fix the known bug in FSW
if filesystem is unavailable, return failure to client requests for RAM

What about other (unknown) bugs that may leave queues disabled?

Added the “Maximum Uptime” (MUT) timer
If awake for more than 36 hours, shutdown

*“when in doubt,
use brute force”
(Ken Thompson)*